**Vectors**

A vector models a dynamic array. Thus, a vector is an abstraction that manages its elements with a dynamic C-style array.

header file

**#include <vector>**

type is defined as a class template inside namespace std:

namespace std {

template <typename T,

typename Allocator = allocator<T> >

class vector;

}

The optional second template parameter defines the memory model. The default memory model is the model allocator.

# Abilities of Vectors

A vector is a kind of ordered collection and provides random access. You can access every element directly in constant time, provided that you know its position. The iterators are random-access iterators, so you can use any algorithm of the STL.

## Size and Capacity

Vectors performance is by allocating more memory than they need to contain all their elements.

Vectors provide the usual size operations size(), empty(), max\_size() and capacity().

The capacity() function returns the number of elements a vector could contain in its actual memory. If you exceed the capacity(), the vector has to reallocate its internal memory.

The capacity of a vector is important for two reasons:

1. Reallocation invalidates all references, pointers, and iterators for elements of the vector.
2. Reallocation takes time.

**Avoid Reallocation**

1. You can use reserve() to ensure a certain capacity before you really need it. In this way, you can ensure that references remain valid as long as the capacity is not exceeded:

std::vector<int> v; // create an empty vector

v.reserve(80); // reserve memory for 80 elements

Calling reserve() with an argument that is less than the current capacity is a no-op.

1. Initialize a vector with enough elements by passing additional arguments to the constructor. For example, if you pass a numeric value as parameter, it is taken as the starting size of the vector:

std::vector<T> v(5); // creates a vector and initializes it with five values (calls five times the default constructor of type T)

C++11 introduced a new member function for vectors: a nonbinding request to shrink the capacity to fit the current number of elements:

v.shrink\_to\_fit(); // request to shrink memory (since C++11)

This request is nonbinding to allow latitude for implementation-specific optimizations. Thus, afterward v.capacity==v.size() may not yield true.

# Vector Operations

## Create, Copy, and Destroy

You can create vectors with and without elements for initialization. If you pass only the size, the elements are created with their default constructor.

|  |  |
| --- | --- |
| Operation | Effect |
| vector<Elem> c | Default constructor; creates an empty vector without any elements |
| vector<Elem> c | (c2) Copy constructor; creates a new vector as a copy of c2 (all elements are copied) |
| vector<Elem> c = c2 | Copy constructor; creates a new vector as a copy of c2 (all elements are copied) |
| vector<Elem> c(rv) | Move constructor; creates a new vector, taking the contents of the rvalue rv (since C++11) |
| vector<Elem> c = rv | Move constructor; creates a new vector, taking the contents of the rvalue rv (since C++11) |
| vector<Elem> c(n) | Creates a vector with n elements created by the default constructor |
| vector<Elem> c(n,elem) | Creates a vector initialized with n copies of element elem |
| vector<Elem> c(beg,end) | Creates a vector initialized with the elements of the range [beg,end) |
| vector<Elem> c(initlist) | Creates a vector initialized with the elements of initializer list initlist (since C++11) |
| vector<Elem> c = initlist | Creates a vector initialized with the elements of initializer list initlist (since C++11) |
| c.~vector() | Destroys all elements and frees the memory |

## Nonmodifying Operations

|  |  |
| --- | --- |
| Operation | Effect |
| c.empty() | Returns whether the container is empty (equivalent to size()==0 but might be faster) |
| c.size() | Returns the current number of elements |
| c.max\_size() | Returns the maximum number of elements possible |
| c.capacity() | Returns the maximum possible number of elements without reallocation |
| c.reserve(num) | Enlarges capacity, if not enough yet |
| c.shrink\_to\_fit() | Request to reduce capacity to fit number of elements (since C++11) |
| c1 == c2 | Returns whether c1 is equal to c2 (calls == for the elements) |
| c1 != c2 | Returns whether c1 is not equal to c2 (equivalent to !(c1==c2)) |
| c1 < c2 | Returns whether c1 is less than c2 |
| c1 > c2 | Returns whether c1 is greater than c2 (equivalent to c2<c1) |
| c1 <= c2 | Returns whether c1 is less than or equal to c2 (equivalent to !(c2<c1)) |
| c1 >= c2 | Returns whether c1 is greater than or equal to c2 (equivalent to !(c1<c2)) |

reserve() and shrink\_to\_fit() manipulate the vector because they invalidate references, pointers, and iterators to elements. However, they are mentioned here because they do not manipulate the logical contents of the container.

## Assignments

|  |  |
| --- | --- |
| Operation | Effect |
| c = c2 | Assigns all elements of c2 to c |
| c = rv | Move assigns all elements of the rvalue rv to c (since C++11) |
| c = initlist | Assigns all elements of the initializer list initlist to c (since C++11) |
| c.assign(n,elem) | Assigns n copies of element elem |
| c.assign(beg,end) | Assigns the elements of the range [beg,end) |
| c.assign(initlist) | Assigns all the elements of the initializer list initlist |
| c1.swap(c2) | Swaps the data of c1 and c2 |
| swap(c1,c2) | Swaps the data of c1 and c2 |

## Element Access

|  |  |
| --- | --- |
| Operation | Effect |
| c[idx] | Returns the element with index idx (no range checking) |
| c.at(idx) | Returns the element with index idx (throws range-error exception if idx is out of range) |
| c.front() | Returns the first element (no check whether a first element exists) |
| c.back() | Returns the last element (no check whether a last element exists) |

For nonconstant vectors, these operations return a reference to the element. Thus, you could modify an element by using one of these operations, provided it is not forbidden for other reasons.

Only at() performs range checking. If the index is out of range, at() throws an out\_of\_range exception.

All other functions do not check. A range error results in undefined behavior.

std::vector<Elem> coll; // empty!

coll[5] = elem; // RUNTIME ERROR ⇒ undefined behavior

std::cout << coll.front(); // RUNTIME ERROR ⇒ undefined behavior

std::vector<Elem> coll; // empty!

if (coll.size() > 5) { coll[5] = elem; } // OK

if (!coll.empty()) { cout << coll.front(); } // OK

coll.at(5) = elem; // throws out\_of\_range exception

## Iterator Functions

Vector iterators are random-access iterators. Thus, in principle you could use all algorithms of the STL.

The exact type of these iterators is implementation defined. For vectors, however, the iterators returned by begin(), cbegin(), end(), and cend() are often ordinary pointers.

If a safe version of the STL that checks range errors and other potential problems is used, the iterator type is usually an auxiliary class.

Iterators remain valid until an element with a smaller index gets inserted or removed or until reallocation occurs and capacity changes.

|  |  |
| --- | --- |
| Operation | Effect |
| c.begin() | Returns a random-access iterator for the first element |
| c.end() | Returns a random-access iterator for the position after the last element |
| c.cbegin() | Returns a constant random-access iterator for the first element (since C++11) |
| c.cend() | Returns a constant random-access iterator for the position after the last element (since C++11) |
| c.rbegin() | Returns a reverse iterator for the first element of a reverse iteration |
| c.rend() | Returns a reverse iterator for the position after the last element of a reverse iteration |
| c.crbegin() | Returns a constant reverse iterator for the first element of a reverse iteration (since C++11) |
| c.crend() | Returns a constant reverse iterator for the position after the last element of a reverse iteration (since C++11) |

## Inserting and Removing Elements

Regarding performance, you should consider that inserting and removing happens faster when:

1. Elements are inserted or removed at the end.
2. The capacity is large enough on entry.
3. Multiple elements are inserted by a single call rather than by multiple calls.

Inserting or removing elements invalidates references, pointers, and iterators that refer to the following elements. An insertion that causes reallocation invalidates all references, iterators, and pointers.

Vectors provide no operation to remove elements directly that have a certain value. You must use an algorithm to do this.

To remove only the first element that has a certain value, you must use the following statements:

std::vector<Elem> coll;

// remove first element with value val

std::vector<Elem>::iterator pos;

pos = find(coll.begin(),coll.end(), val);

if(pos != coll.end()) { coll.erase(pos); }

|  |  |
| --- | --- |
| Operation | Effect |
| c.push\_back(elem) | Appends a copy of elem at the end |
| c.pop\_back() | Removes the last element (does not return it) |
| c.insert(pos, elem) | Inserts a copy of elem before iterator position pos and returns the position of the new element |
| c.insert(pos, n, elem) | Inserts n copies of elem before iterator position pos and returns the position of the first new element (or pos if there is no new element) |
| c.insert(pos, beg, end) | Inserts a copy of all elements of the range [beg,end) before iterator position pos and returns the position of the first new element (or pos if there is no new element) |
| c.insert(pos, initlist) | Inserts a copy of all elements of the initializer list initlist before iterator position pos and returns the position of the first new element (or pos if there is no new element; since C++11) |
| c.emplace(pos, args...) | Inserts a copy of an element initialized with args before iterator position pos and returns the position of the new element (since C++11) |
| c.emplace\_back(args...) | Appends a copy of an element initialized with args at the end (returns nothing; since C++11) |
| c.erase(pos) | Removes the element at iterator position pos and returns the position of the next element |
| c.erase(beg, end) | Removes all elements of the range [beg,end) and returns the position of the next element |
| c.resize(num) | Changes the number of elements to num (if size() grows new elements are created by their default constructor) |
| c.resize(num, elem) | Changes the number of elements to num (if size() grows new elements are copies of elem) |
| c.clear() | Removes all elements (empties the container) |

# Using Vectors as C-Style Arrays

C++ standard library guarantees that the elements of a vector are in contiguous memory. Thus, you can expect that for any valid index i in vector v, the following yields true:

&v[i] == &v[0] + i

It simply means that you can use a vector in all cases in which you could use a dynamic array. For example, you can use a vector to hold data of ordinary C-strings of type char\* or const char\*:

std::vector<char> v; // create vector as dynamic array of chars

v.resize(41); // make room for 41 characters (including ’\0’)

strcpy(&v[0],"hello, world"); // copy a C-string into the vector

printf("%s\n", &v[0]); // print contents of the vector as C-string

## data()

Since C++11, you don’t have to use the expression &a[0] to get direct access to the elements in the vector, because the member function data() is provided for this purpose:

std::vector<char,41> v; // create static array of 41 chars

strcpy(v.data(),"hello, world"); // copy a C-string into the array

printf("%s\n", v.data()); // print contents of the array as C-string

Do not pass an iterator as the address of the first element. Iterators of vectors have an implementation-specific type, which may be totally different from an ordinary pointer:

printf("%s\n", v.begin()); // ERROR (might work, but not portable)

printf("%s\n", v.data()); // OK (since C++11)

printf("%s\n", &v[0]); // OK, but data() is better

# Exception Handling

at() is the only member function which may throw an exception

If functions called by a vector throw exceptions, the C++ standard library provides the following guarantees:

1. If an element gets inserted with push\_back() and an exception occurs, this function has no effect.
2. insert(), emplace(), emplace\_back(), and push\_back() either succeed or have no effect, provided that the copy/move operations (constructors and assignment operators) of the elements do not throw.
3. pop\_back() does not throw any exceptions.
4. erase() does not throw if the copy/move operations (constructors and assignment operators) of the elements do not throw.
5. swap() and clear() do not throw.
6. If elements are used that never throw exceptions on copy/move operations (constructors and assignment operators), every operation is either successful or has no effect. Such elements might be “plain old data” (POD). POD describes types that use no special C++ feature. For example, every ordinary C structure is POD.

All these guarantees are based on the requirements that destructors don’t throw.

# END